

(19)日本国特許庁 (J P)

(12) 公 開 特 許 公 報 (A)

(11)特許出願公開番号

特開平8-263299

(43)公開日 平成8年(1996)10月11日

(51)Int.Cl. <sup>a</sup>	識別記号	庁内整理番号	F I	技術表示箇所
G 0 6 F 9/45		7737-5B	G 0 6 F 9/44	3 2 2 F
9/06	5 4 0		9/06	5 4 0 A

審査請求 未請求 請求項の数 6 O L (全 9 頁)

(21)出願番号 特願平7-67298

(22)出願日 平成7年(1995)3月27日

(71)出願人 000001889

三洋電機株式会社

大阪府守口市京阪本通2丁目5番5号

(72)発明者 甲村 康人

大阪府守口市京阪本通2丁目5番5号 三洋電機株式会社内

(72)発明者 三浦 宏喜

大阪府守口市京阪本通2丁目5番5号 三洋電機株式会社内

(72)発明者 松本 健志

大阪府守口市京阪本通2丁目5番5号 三洋電機株式会社内

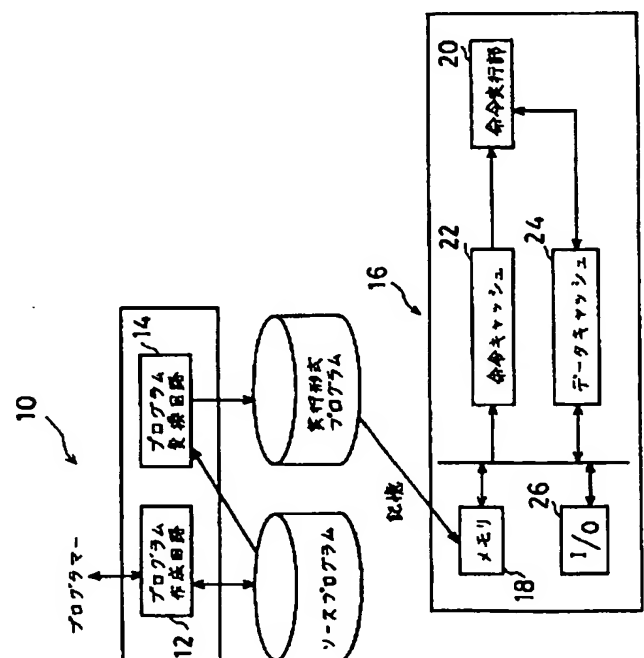
(74)代理人 弁理士 山田 義人

(54)【発明の名称】 プログラム変換方法

(57)【要約】

【構成】 プログラマによってプログラム作成回路12で作成されたソースプログラムは、プログラム支援回路14で以下のようにして実行形式プログラムに変換され、このプログラムがデータ処理装置16で実行される。すなわち、まずプログラムを複数の部分プログラムに分割し、次に各部分プログラムどうしのパターンマッチングをとることによって、部分プログラムの中から機能的に類似している部分プログラムを抽出する。続いて、抽出した部分プログラムのそれぞれの機能を解析して、共用サブルーチンを生成する。その後、先ほど抽出した部分プログラムのそれぞれを共用サブルーチンの呼び出し命令に置き換える。

【効果】 互いに類似している部分プログラムが、共用サブルーチンに置き換わるので、ソースコードレベルでの書き換えを必要とせずに、全体のプログラムコード量を削減することができる。



## 【特許請求の範囲】

【請求項 1】(a) プログラムを複数の部分プログラムに分割し、

(b) 前記部分プログラムどうしのパターンマッチングをとることによって前記部分プログラムの中から機能的に類似する部分プログラムを抽出し、

(c) 前記類似する部分プログラムのそれぞれの機能を解析することによって前記機能の全てを満たす共用サブルーチンを生成し、そして

(d) 前記類似する部分プログラムのそれぞれを前記共用サブルーチンの呼び出し命令に置換する、プログラム変換方法。

【請求項 2】(e) 前記ステップ(a) に先立って高級言語で記述されたソースプログラムを中間言語プログラムに変換し、そして

(f) 前記ステップ(d) に続いて前記ステップ(c) で生成された共用サブルーチンおよび前記ステップ(d) で置換された中間言語プログラムに基づいて実行形式プログラムを生成する、請求項 1 記載のプログラム変換方法。

【請求項 3】前記中間言語プログラムの記述形式として高級言語のコンパイルの過程で生成されるレジスタ転送レベルの記述形式を用いる、請求項 2 記載のプログラム変換方法。

【請求項 4】前記中間言語プログラムの記述形式として高級言語のコンパイルの過程で生成される構文木レベルの記述形式を用いる、請求項 2 記載のプログラム変換方法。

【請求項 5】前記ステップ(a) は前記プログラムを  $j$  個 ( $j \geq 1$ ) の連続した基本ブロックを分割単位として複数の部分プログラムに分割し、前記ステップ(b) は部分プログラムの間でパターンマッチングをとる、請求項 1 ないし 4 のいずれかに記載のプログラム変換方法。

【請求項 6】前記ステップ(a) は、(a-1)  $j$  が所定値になるまで前記  $j$  をインクリメントし、(a-2) 前記  $j$  が変化する都度前記プログラムを各々が  $j$  個の連続した基本ブロックを含む複数の部分プログラムに分割し、前記ステップ(b) は前記  $j$  が前記所定値に達したときパターンマッチングを実行する、請求項 5 記載のプログラム変換方法。

## 【発明の詳細な説明】

## 【0001】

【産業上の利用分野】この発明はプログラム変換方法に関し、特にたとえばプログラム作成者が作成したソースプログラムを特定のデータ処理装置の実行形式プログラムに変換する、プログラム変換方法に関する。

## 【0002】

【従来の技術】マイクロプロセッサなどのデータ処理装置は、メモリに記憶されたプログラムを読み込み実行するが、近年のエレクトロニクス製品の高機能化ニーズに伴い、データ処理装置が実行するプログラムのサイズは急

激に増大している。このため、データ処理装置において、中央処理装置(CPU)のハードウェアコストよりも、プログラムメモリのハードウェアコストの方が遥かに大きくなるというケースが増えており、プログラムのサイズを小さくすることが重要かつ重大な課題となっている。特に、プログラム作成者が、C言語などの高級言語によってプログラムを作成する場合には、アセンブリ言語によりプログラムを作成する場合に比して、実行形式のプログラムコード量がさらに大きくなるという状況であった。

【0003】従来より、このような問題を解決するために、高級言語コンパイラにより、プログラム・コードに対する様々な最適化が行われてきたが、大幅なコード量の削減には至らなかった。

## 【0004】

【発明が解決しようとする課題】大規模なプログラムを高級言語やアセンブリ言語などによって記述する場合、プログラマは通常、プログラムの仕様に即して、各部において遂行したい機能に即して、各部分のプログラムを記述していく。このため、プログラム全体においては、結果的に、類似の機能をもつプログラム部分が各所に存在する場合が多い。コンパイラやアセンブラは、このような類似のプログラム部分を多数含むプログラムをそのままコンパイル、あるいはアセンブルするため、結果的に、実行形式プログラムのコードサイズが大きくなるという問題があった。

【0005】しかしながら、プログラマが、既に記述した全てのプログラム部分の機能と処理内容を常時把握し、類似の部分を発見して、これを解析し、サブルーチンなどの共用プログラムに置き換えていくことはプログラマの大きな負担となり、非常に困難な作業である。また、このような置き換えをソースコードレベルで行うことは、プログラムそのものの了解性や保守性も損なうことにつながるため、得策ではない。

【0006】それゆえに、この発明の主たる目的は、プログラマの作業を軽減するとともに、ソースコードレベルでのプログラムの書き換えを必要とせず、実行コードサイズを削減することのできる、プログラム変換方法を提供することである。

## 【0007】

【課題を解決するための手段】この発明は、(a) プログラムを複数の部分プログラムに分割し、(b) 部分プログラムどうしのパターンマッチングをとることによって部分プログラムの中から機能的に類似する部分プログラムを抽出し、(c) 類似する部分プログラムのそれぞれの機能を解析することによって機能の全てを満たす共用サブルーチンを生成し、そして(d) 類似する部分プログラムのそれぞれを共用サブルーチンの呼び出し命令に置換する、プログラム変換方法である。

## 【0008】

【作用】まずプログラムを複数の部分プログラムに分割し、次に各部分プログラムどうしでパターンマッチングを実行することによって、部分プログラムの中から機能的に類似している部分プログラムを抽出する。続いて、抽出した部分プログラムのそれぞれの機能を解析することによって、それら全ての機能を満たすような共用サブルーチンを生成する。その後、先ほど抽出した部分プログラムのそれぞれを共用サブルーチンの呼び出し命令に置き換える。

【0009】

【発明の効果】この発明によれば、互いに類似している部分プログラムが、「サブルーチンの呼び出し」という簡単で命令数の少ない記述と共用サブルーチンとに置き換わるので、ソースコードレベルでの書き換えを必要とせずに、全体のプログラムコード量を削減することができる。

【0010】この発明の上述の目的、その他の目的、特徴および利点は、図面を参照して行う以下の実施例の詳細な説明から一層明らかとなろう。

【0011】

【実施例】図1を参照して、この実施例のプログラム開発支援装置10は、プログラム作成回路12を含む。プログラマは、このプログラム作成回路12によってソースプログラムを作成し、その後プログラム変換回路14によってソースプログラムを実行形式プログラムに変換する。このようにして開発された実行形式プログラムは、データ処理装置16に含まれるメモリ18に書き込まれ、命令実行部20で実行形式プログラムが実行される。すなわち、命令実行部20は、命令キャッシュ22によってメモリ18から実行形式プログラムを読み出すとともに、命令実行に必要なデータをデータキャッシュ24によってメモリ18およびI/O装置26から読み出し、実行する。

【0012】プログラム変換回路12はコンパイラであって、図2および図3に示すアルゴリズムに従ってソースプログラムを実行形式プログラムに変換する。すなわち、まず図2のステップS1においてソースプログラムを構文解析し、次にステップS3において中間言語（レジスタ転送記述）のプログラムを生成する。続いて、ステップS5およびS7のそれぞれにおいて高レベル最適化、およびデータフロー解析をし、その後ステップS9において、パターンマッチングによって互いに類似する部分プログラムを抽出するとともに抽出した個所を共用サブルーチンに置き換える。続いて、ステップS11において、変換された中間言語プログラムに含まれる命令をスケジューリングし、ステップS13においてレジスタを割り当てた後、ステップS15において実行形式プログラムを生成する。

【0013】ステップS9は具体的には図3および図4に示すアルゴリズムによって処理される。すなわち、ま

ずステップS21において、中間言語プログラムを読み込む。次にステップS23において、この中間言語プログラムをj個の基本ブロック（中間言語プログラム中の分岐命令の飛び先となる位置（ラベル位置）にある命令から次のラベル位置の直前にある命令まで）を含む複数の部分プログラムに分割し、各部分プログラム間でパターンマッチングを実行し、類似の部分プログラム分を抽出する。続いて、ステップS25およびS27のそれぞれにおいて、各部分プログラムの共通部分および差異部分を抽出し、ステップS29において共用サブルーチンを生成する。その後、ステップS31において類似の部分プログラムの各々を共用サブルーチンで置き換え、ステップS33において類似の部分プログラム分がまだあるか否か判断する。そして、“YES”であればステップS25に戻るが、“NO”であれば図2のステップS11に移行する。

【0014】ステップS23は、具体的には図4に示すようなアルゴリズムとなっている。すなわち、まずステップS41においてjを定数Sに設定するとともにkをゼロに設定し、次にステップS43においてmをゼロに設定する。続いて、ステップS45において中間言語プログラムからラベル位置を検出し、ステップS47において基本ブロックを検出する。基本ブロックが検出されると、ステップS49においてmをインクリメントし、ステップS51においてm=jであるか否か判断する。ここで、m=jでなければ、ステップS45に戻り、再び基本ブロックの検出を開始するが、m=jであれば、ステップS53においてここまでを部分プログラムとし、ここで中間言語プログラムを分割する。続いて、ステップS55においてkをインクリメントし、ステップS57において中間言語プログラムの終わりまで来たか否か判断する。そして、“NO”であればステップS43に戻り、再び部分プログラムの生成を開始するが、“YES”であれば、ステップS59において、生成された全ての部分プログラム間でk×(k-1)回のパターンマッチングを実行する。その後、ステップS61において互いに類似する部分プログラムを抽出し、図3のステップS25に移行する。

【0015】このようにプログラム変換回路12が処理することによって、ソースプログラムは図4に示すような手順で実行形式プログラムに変換される。すなわち、たとえばプログラマがプログラム作成回路によって高級言語で記述したソースプログラムは、プログラム変換過程において中間言語プログラムに変換される。次に、中間言語プログラムから、機能的に類似した複数の部分プログラムが抽出され、抽出された部分プログラムから、それらの機能を満たす共用サブルーチンが生成される。続いて、中間言語プログラムの互いに類似する部分プログラム個所が共用サブルーチン呼び出し指令に置き換えられ、新たな中間言語プログラムが生成される。その

10

20

30

40

50

後、このプログラム変換処理後の中間言語プログラムが、実行形式のプログラムに変換される。

【0016】図5～図8を参照して、C言語を用いたソースプログラムに基づいて実行形式プログラムを作成する動作について説明する。図5に示すソースプログラムは、メインの関数から  $f()$  と  $g()$  の2つの関数を呼び出しているものである。このプログラムを構文解析すると、図6に示す中間言語プログラムが生成される。そして、この中間言語プログラムから機能的に類似した複数の部分プログラムが抽出される。すなわち、中間言語プログラムが複数の部分プログラムに分割され、各部分プログラムどうしのパターンマッチングがなされた後、複数の部分プログラムの中から機能的に類似している部分プログラム（ブロック1およびブロック2）が抽出される。

【0017】次に、抽出した類似の部分プログラムの各々の機能を解析することによって、それら全ての機能を満たすような共用サブルーチンが生成される。図6においては上述のようにブロック1とブロック2とが類似するため、共用サブルーチンとして図7のブロック5が生成される。そして、図6のブロック1およびブロック2が、図7のブロック3およびブロック4に示すように、共用サブルーチン（ブロック5）の呼び出しに置き換えられる。

【0018】このように生成された中間言語プログラムに対して、さらに、不要なサブルーチンの呼び出しやサブルーチンからの復帰が削除され、これによって、図8に示すようなプログラム変換後の中間言語プログラム58が生成される。すなわち、図7において、関数  $f()$  および  $g()$  は、サブルーチン呼び出しのための引き数の設定およびサブルーチンの呼び出しのみになっているため、図8の中間言語プログラムに変換される。プログラム変換された中間言語プログラムは、その後、命令スケジューリングおよびレジスタ割り当てを経て、実行形式プログラムに変換される。

【0019】この実施例によれば、ソースプログラムから実行形式のプログラムを生成する過程で作られる中間言語プログラムにおいて、パターンマッチングにより機能的に類似のブロックをサブルーチンおよびサブルーチンの呼び出しに置き換えることで、プログラムが小さくなる。したがって、実行形式プログラムのコード量を削減でき、データ処理装置のメモリ量をより小さなものとすることができる。

【0020】なお、この実施例では、連続する  $S$  個の基本ブロックを結合することによって部分プログラムを生成し、この部分プログラムから互いに類似する部分プログラムを抽出して共用サブルーチンを生成するようにしたが、 $S \leq j \leq Q$  を満たすそれぞれの整数だけ基本ブロックが連続する部分プログラムを生成し、生成された全ての部分プログラムの間でパターンマッチングをとるこ

とによって共用サブルーチンを生成するようにしてもよい。このような場合、プログラム変換回路12は図3に示すステップS23を図10に示すアルゴリズムに従って処理する必要がある。

【0021】すなわち、まずステップS71において  $j$  を定数  $S$  に設定するとともに、 $j$  の最大値  $j_{max}$  を定数  $Q$  ( $Q > S$ ) に設定する。次に、ステップS37において  $k=0$  とし、ステップS75～S89に示す処理をする。なお、これらのステップは上述のステップS43～S57と同様であるので、重複した説明を省略する。ステップS89において“YES”であれば、ステップS91において  $k$  個の部分プログラムをリストアップし、ステップS93において  $j$  をインクリメントする。その後、ステップS95において  $j > j_{max}$  であるか否か判断し、“NO”であればステップS73に戻るが、“YES”であれば、ステップS97においてリストアップされた全ての部分プログラム間でパターンマッチングを実行し、ステップS99において互いに類似する部分プログラムを抽出する。ここで、全ての部分プログラム間でパターンマッチングをとると、互いに共有部分を有する部分プログラム間でもパターンマッチングがとられるが、共有部分を有するプログラムの間では基本ブロック数が異なるため、互いがマッチすることはなく、何ら問題は無い。

【0022】なお、この実施例ではソースプログラムの記述にC言語を用いているが、この発明はこの場合に限らず、ソースプログラムが他の高級言語で記述される場合にも適用できることはもちろんである。このような場合、中間言語プログラムについて共用サブルーチンが生成されることは限らないことはもちろんである。

【図面の簡単な説明】

【図1】この発明の一実施例を示すブロック図である。

【図2】図1実施例の動作の一部を示すフロー図である。

【図3】図1実施例の動作の一部を示すフロー図である。

【図4】図1実施例の動作の一部を示すフロー図である。

【図5】図1実施例の動作の一部を示す図解図である。

【図6】この発明の一実施例を示すプログラムである。

【図7】この発明の一実施例を示すプログラムである。

【図8】この発明の一実施例を示すプログラムである。

【図9】この発明の一実施例を示すプログラムである。

【図10】他の実施例の動作の一部を示すフロー図である。

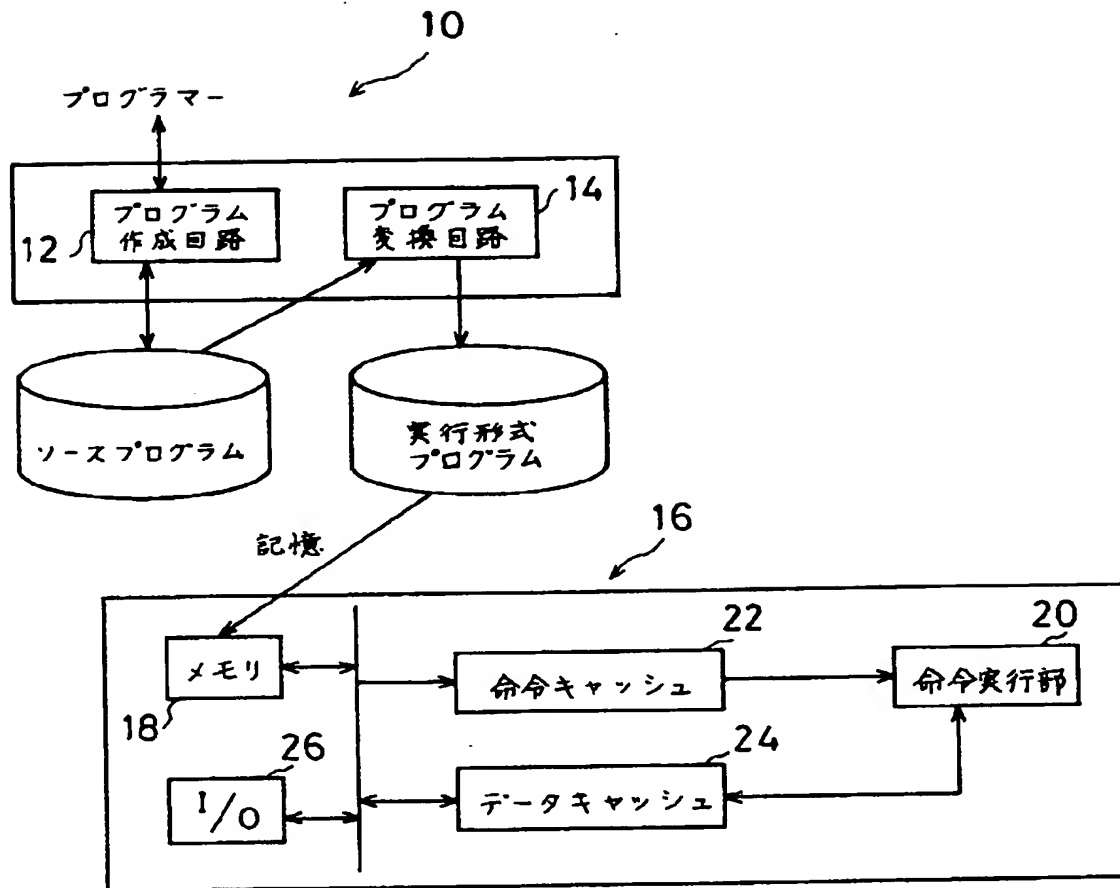
【符号の説明】

10 …プログラム開発支援装置

14 …プログラム変換回路

16 …データ処理装置

【図1】



【図9】

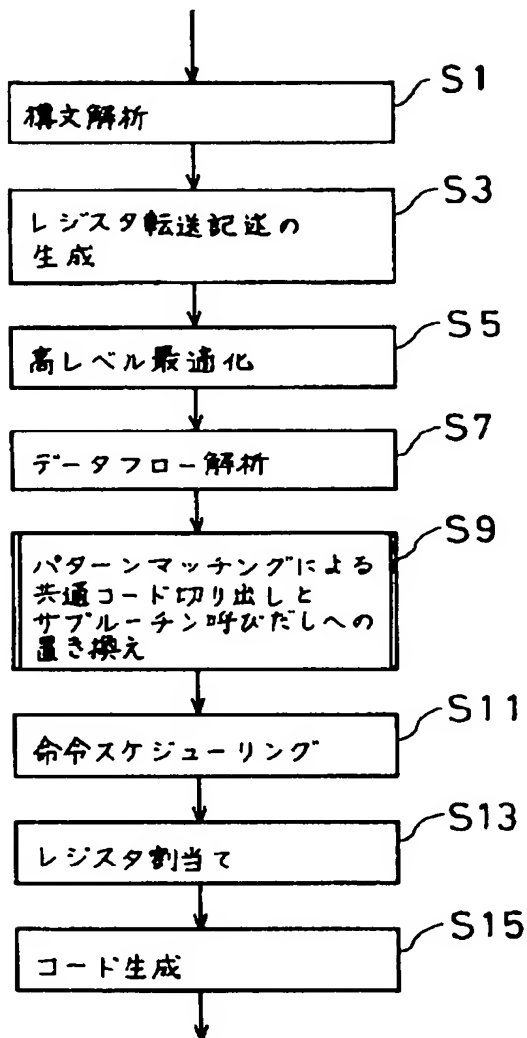
```

;;; main の RTL
引数[0] := 100
call generated_subr_1
引数[0] := 50
call generated_subr_1
return

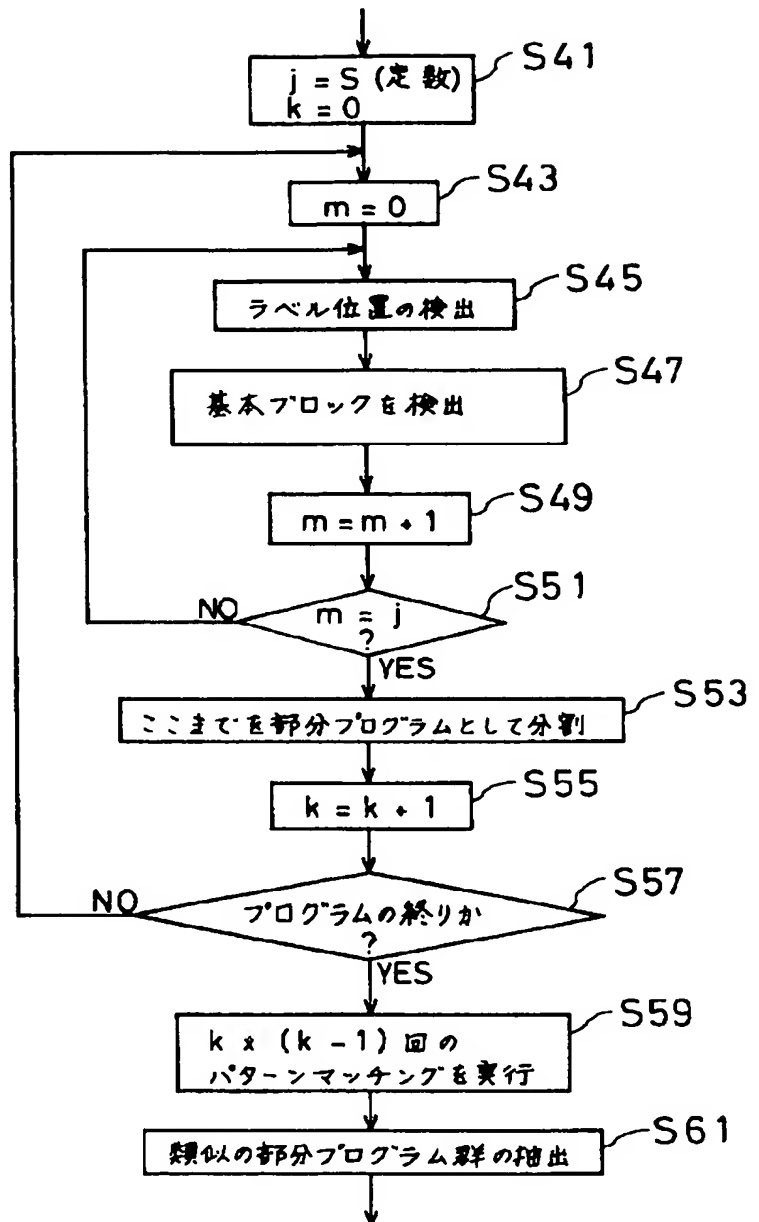
;;; generated_subr_1 の RTL
r93 := 引数[0]
r95 := 0
r94 := 0
if r95 >= r93 then L36
L21:
r95 := r95 + r94
r94 := r94 + 1
if r94 < r93 then L21
L36:
返り値 := r95
return

```

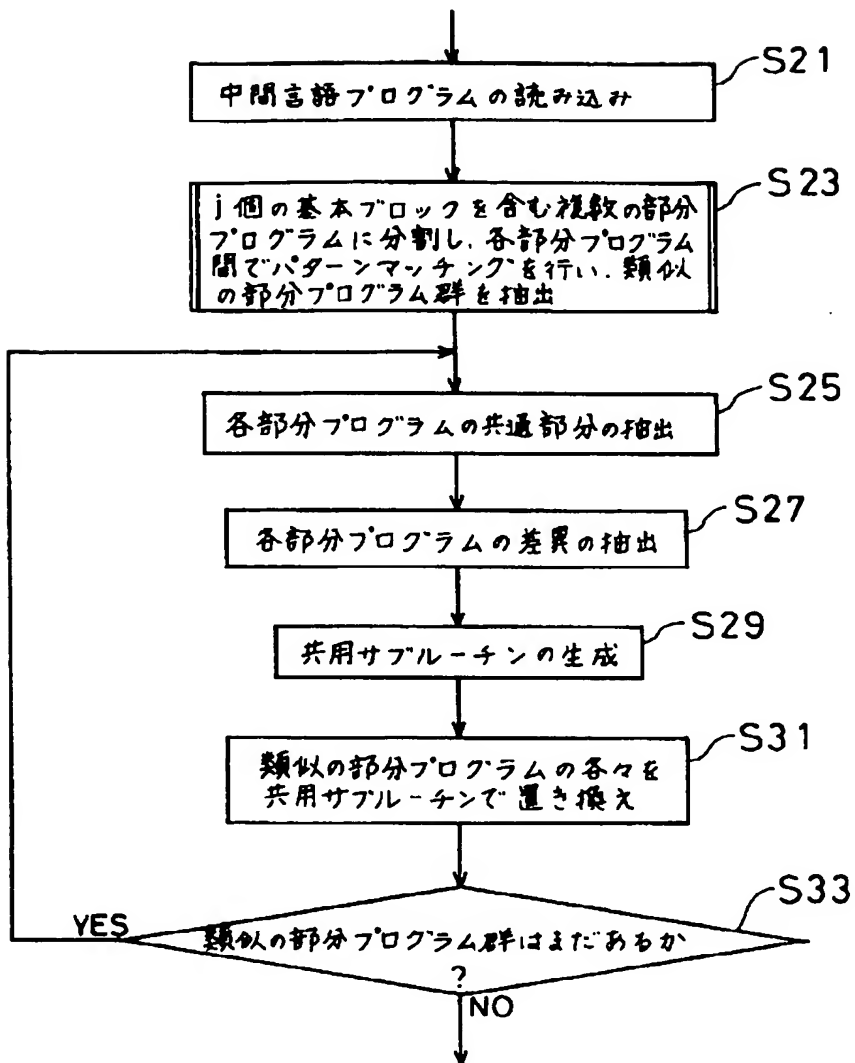
【図2】



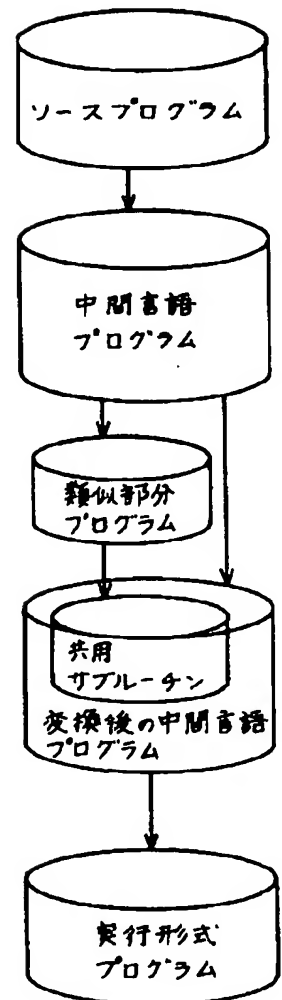
【図4】



【図3】



【図5】



【図6】

```

main()
{
    f(100);
    g();
}

f(n)
{
    int n;
    int i, r;

    r = 0;
    for (i = 0; i < n; i++) {
        r += i;
    }
    return r;
}

g()
{
    int i, r;

    r = 0;
    for (i = 0; i < 50; i++) {
        r += i;
    }
    return r;
}

```

【図7】

```

;;; main の RTL
引数[0] := 100
call f
call g
return

;;; f の RTL
r93 := 引数[0]
r95 := 0
r94 := 0
if r95 >= r93 then L136
L121:
r95 := r95 + r94
r94 := r94 + 1
if r94 < r93 then L121
L136:
    返り値 := r95
return

;;; g の RTL
r94 := 0
r93 := 0
r95 := 49
L221:
r94 := r94 + r93
r93 := r93 + 1
if r93 <= r95 then L221
    返り値 := r94
return

```

ブロック 1

ブロック 2

【図8】

```

;;; main の RTL
引数[0] := 100
call f
call g
return

;;; f の RTL
call generated_subr_1      …… ブロック 3
return

;;; g の RTL
引数[0] := 50
call generated_subr_1      …… ブロック 4
return

;;; generated_subr_1 の RTL
r93 := 引数[0]
r95 := 0
r94 := 0
if r95 >= r93 then L36
L21:
r95 := r95 + r94
r94 := r94 + 1
if r94 < r93 then L21
L36:
    返り値 := r95
return

```

ブロック 5



【図10】

